On the use of graph transformations for model composition traceability

Youness Laghouaouta, Adil Anwar, Mahmoud Nassar, Jean-Michel Bruel

On the Use of Graph Transformations for Model Composition Traceability

Youness Laghouaouta
IMS-SIME, Mohamed Vth Souissi University
Rabat, Morocco
y.laghouaouta@um5s.net.ma

Adil Anwar
Siweb, EMI, Mohamed Vth Agdal University
Rabat, Morocco
anwar@emi.ac.ma

Mahmoud Nassar
ENSIAS, Mohamed Vth Souissi University
Rabat, Morocco
nassar@ensias.ma

Jean-Michel Bruel
IRIT/Université de Toulouse
Toulouse, France
bruel@irit.fr

***Abstract*—The model composition provides support to build systems based on a set of less complex sub-models. This operation allows managing complexity while supporting the modularity and reusability tasks. Due to the increase number of the involving models, their composition becomes a tedious task. For that, the need for maintaining traceability information is raised to help managing the composition operation. We propose in this work a graph-based model transformations approach, which aims to keep track of the model composition operation. Our objective is to capture traces in an automatic and reusable manner. Finally, a composition scenario is given to demonstrate the feasibility of our proposal.**

Keywords*—*model traceability, model composition, aspect-oriented modeling, graph transformations.

I. Introduction

Large and heterogeneous systems are too complex to be described using a single model. Indeed, the model composition provides support to build systems based on different models. Each one refers to a specific concern, perspective, point of view or component [1]. Such modularization allows managing the system complexity by reasoning about less complex sub-models.

Despite of the model composition benefits such as: validation of involving models and models synchronization; this operation remains a laborious and error prone activity. We argue that a traceability mechanism is a key factor to handle this task. Traces help designer to comprehend the exact effects of the composition and reveal the interactions among involving models. Besides, such information provides means to validate the composition and assist the propagation of changes during the evolution of the system.

In this context, we propose a traceability approach for the model composition operation. Within our proposal, we consider the traceability management as a crosscutting concern, while the weaving of the traces generation patterns is specified using graph transformations [2]. The incorporated structure allow the generation of the trace model as an additional target model. Such an extra output model has manifold application in the model composition field:

- *To validate the composition*: trace links provide a detailed view of the flow of execution. Indeed, they represent relationships between source model elements and their target equivalents. Through these links, we can verify the consistency and the completeness of the model composition operation.
- *To support co-evolution of models*: the trace model specifies how source artifacts participate in the production of the composed model. Those links are useful to analyze the impact of changing sub-models during the evolution of the system.
- *To optimize the composition chain*: As part of a model composition chain, the restriction to source artifacts of a given stage of the overall chain confuses its management. Hence, the use of the trace model can broaden its scope, through the reuse of previous valuable links.

The remained of this paper is structured as follows: Section II presents a review on the traceability management and a discussion about the need of a traceability mechanism addressing the model composition operation. In Section III we propose an overview of our approach. Thereafter, in Section IV we implement our traceability aspect for the ATL language; while Section V presents a case study that demonstrates the applicability of our proposal. In Section VI we review the related approaches. Finally Section VII summaries this paper and represents future works.

II. Background and motivation

The IEEE Standard Glossary of Software Engineering Terminology [3] defines traceability as:

"*the degree to which relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one other; for example, the degree to which the requirements and design of a given software component match*"

The definition is tied to the abstraction level of the managed artifacts and the traceability scenario. Indeed, it is closely

related to the requirements traceability field. As regard to model transformations, traceability is defined as:

- "*Any relationship that exists between artifacts involved in the software engineering life cycle*" [4].
- "*The runtime footprint of model transformation .Essentially, trace links provide this kind of information by associating source and target model element with re-spect of the execution of a certain model transformation*" [5].

Essentially, traceability refers to the ability to manage links between elements handled by a model management operation. It provides a view of the changes that have occurred in these model elements and reveals the complexity of logical relations [6] existing among them.

Trace links of a model transformation can either be generated naturally with an internal traceability tool (implicit traceability) or produced using an external support (explicit traceability). In the former case, the generation task does not require an additional effort; however, the generation process is fixed and cannot be configured to produce the required traces with respect to a given traceability scenario. In addition, the structuring of traces is often simple to allow an advanced post configuration. As for the implicit traceability, even if there is a need for encoding the traces generation; but it provides manifold configuration tracks. Indeed, both the traceability metamodel and the application scope can be chosen freely by the developer.

An outline of the grand challenges concerning the traceability management has been stated by the Center of Excellence for Software Traceability [7]. The first challenge is the purpose of generating traces. Indeed, the traceability concern has to fit the user's intensions. In addition, the authors argue that traces have to be configurable, portable and generated in a scalable manner. In order to satisfy these requirements, we propose a traceability approach for the model composition operation. Actually, we have found in the literature various traceability solutions that address the model transformation field; however, we have not encountered a specific approach concerning the composition operation. Therefore, adopting such solutions does not deal with the purpose challenge, because the composition has particular intensions and the traceability support has to be aware of them. On the other hand, we believe that existing solutions do not provide means to express configurable traces, since they are designed in such a way that disregards the composition process. Our objective is to gather the benefits of the existing model transformation traceability solutions while focusing on the model composition field. In fact, we have set the traceability requirements below according to the analysis we have conducted of the main model transformation traceability approaches [8]:

- The traceability data has to be stored in a separate model in order to keep the managed models clean. Besides, this trace model has to be expressed using a generic metamodel to reduce effort to achieve traceability. This mechanism allows the reusability of traces.
- The traceability metamodel must provide an extensibility mechanism for expressing highly configurable traces with respect to the traceability purpose and specifications of contributing models.
- Regarding the scalability challenge, we have to generate traces with an automatic mechanism. Besides, the generation process must be configurable with respect to the traceability scenario.

Finally, we have taken into account the fact that our proposal must support a visualization system for expressing the trace model in a user friendly representation.

III. Overview of the approach

A. Traces generation process

In this section we provide an overview of the way traces are captured and structured. We aim through our approach keep track of the model composition operation. For that, we propose to generate the trace model as being an additional target model of the specification to trace (see Fig. 1). We consider the traceability concern as being a cross-cutting concern. Indeed, we define a traceability aspect that automatically weaves the traces generation patterns. Traces are conforming to a generic traceability metamodel that deals with the configuration and portability challenges.

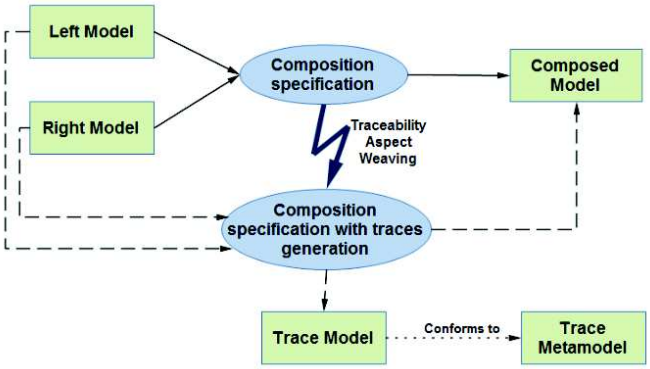

Fig. 1. Traceability generation process

B. Traces structuring

we presents in Fig. 2 the traceability metamodel accounting for structuring of traces. This formalism allows expressing the composition relationship kinds in a trivial manner. The composition operation has a particular process [9]. It consists of detecting matching elements that describe the same concept in the left and right models. These pairs of elements are merged while other elements are eventually transformed into the target model. Hence, the structuring of traces must take account of these specificities in order to express purposed links. Accordingly, we have set two types of trace links: merging and transformation links.

- *TraceLink*: generalizes the relationship between the source and target model elements handled by the model composition operation.
- *MergingLink*: connects the left and right model elements to the merged ones.

- *TransformationLink*: expresses a transition from a source model element (belonging to the left or right model) to their target equivalents.
- *ModelElement*: This concept refers the linked model elements.
- *Context*: allows expressing semantically rich traces through the assignment of further information to a sub-set of traces. This extensibility mechanism is based on the definition of the relevant context attributes.
- *ContextAttribute*: represents the additional information to be assigned to a sub-set of trace links, such as: the intension of capturing those traces, the rule that generates them...
- *TraceModel*: it is the root element which contains all the generated trace links and contexts.

Besides, we represent the rule invocation through parent-child relations among trace links. This provides a multi-scales character to the trace model, and allows the user to examine traces at different granularity degree.

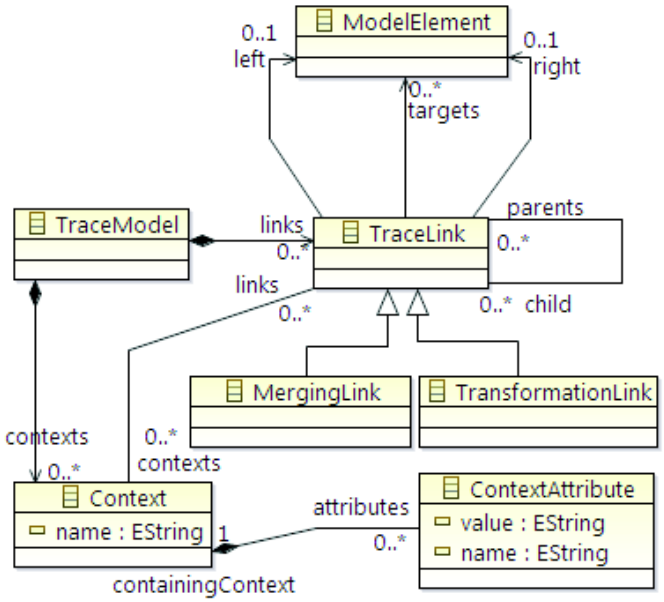

Fig. 2. Composition traceability metamodel

C. Traces generation

Aspect Oriented Modeling (AOM) [10] applies aspect oriented programming [11] in the context of model driven engineering. It focuses on modularizing and composing cross-cutting concerns during the design phase of a software system. Aspect oriented approaches aim at separating crosscutting concerns (security, persistence...) from the base concerns of a system. In AOM, both the aspects that encapsulate the crosscutting structures and the base model they crosscut are models. An aspect is defined principally by:

- *A pointcut*: it is a predicate over a model used to determine the places where the aspect should be applied (*joinpoints*).
- *An advice*: it is the new structure that replaces the relevant jointpoints.

Within our approach, we consider traceability as a cross-cutting concern to deal with the generation task while supporting the portability and scalability challenges. Indeed, the traceability concern is encapsulated in a reusable aspect that is used to automatically weave the traces generation patterns. Furthermore, AOM provides support to keep track of the composition operation regardless the concrete syntax of the composition language (textual or graphical) by abstracting the specification through its corresponding model.

The weaving mechanism is implemented using graph transformations. A graph transformation rule consists of two parts, a left-hand side (LHS) and a right-hand side (RHS). A rule is applied by replacing the objects of the left-hand side with the objects of the right-hand side, only if the pattern of the left-hand side can be matched to a given graph [12]. In what follows, the traceability aspect corresponds to a set of graph transformation rules. The LHS parts determine the places where the aspect should be applied (**joinpoints**) and the RHS parts define the new structures that replace the relevant joinpoints (**advice**).

The graph transformation unit depicted in Fig. 3 presents an overview of the weaving process. The *TraceModelDeclaration* rule allows declaring the trace model to be an additional target model. Thereafter, the second rule is applied to trace composition rules. The tracing of a rule consists of providing it with the behavior it needs to produce a trace link that connects the source elements with the composed ones. Basically, this is based on the declaration of an additional output element that refers the traceability link and the assignment of the traceability data (*left*, *right* and *targets* reference) to it (see Fig.2). Subsequently, the rule *TraceLinksNesting* weaves the patterns responsible of the nesting of traces with respect to the rule calls. Finally, the last rule deletes all the temporary information we use to perform the weaving process. The next section illustrates the application of this weaving process to trace a specification written in the ATL language [13].

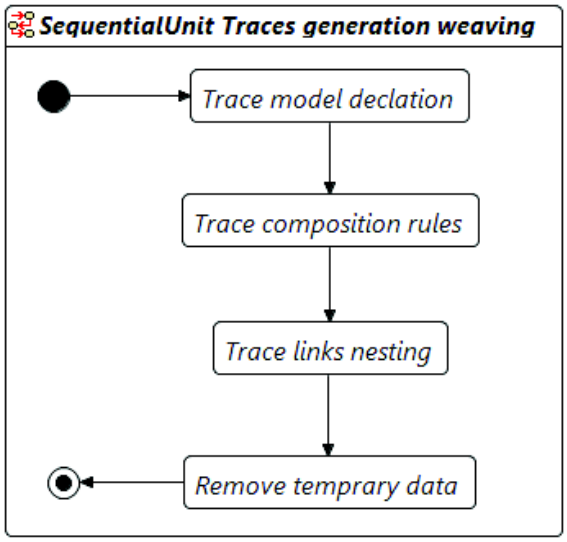

Fig. 3. Traces generation weaving unit

IV. Application of Graph Transformations for Traces Generation

In this section, we apply the traceability weaving process for the ATL language. ATL is a model transformation language that provides tools to compute a set of target models from source models. Nevertheless, it does not support means to express the composition scenario in a native manner. We have experienced the applicability of our solution to trace composition oriented approaches like EML [14] and amar et al. [1]. By choosing the ATL language, we aim to reveal the generic character of our proposal and present the way we master the specificities of a given language using AOM.

A. Traceability management within ATL

Jouault [15] proposed a traceability approach to overcome the limitation of the implicit traceability in ATL. It addresses two configuration dimensions: the *range* and the *format*. The range refers to the possibility to select a sub-set of elements to trace, while the format corresponds to the traces structuring. The generation of trace links is based on a Higher-Order Transformation (HOT) named *TraceAdder*. This HOT automatically inserts the traces generation code to any existing ATL program.

Yie and Wagelaar [16] presented a solution that enlarges the limited access to the implicit links which is handled by the *resolveTemp* operation. They proposed two mechanisms to provide this rich access. The first one consists of an extension of the ATL virtual machine to provide the selection of the target elements by type and copying the implicit *transient links* into an external trace model. Furthermore, they present another mechanism that allows generating the trace model on demand through byte code adaptation.

Although the proposal of the aforementioned approaches is structured around the configuration issue, they use very simple traceability metamodels which are not amenable to generate highly configurable trace models. Actually, the authors have not proposed way for adding customized tracing information or for adapting their generation process to other traceability metamodel mean to express valuable and interesting traces depending on the traceability context which is the model composition in our case.

B. Traces generation for ATL

We have chosen the ATL language as an example of transformation language used to specify the merging scenarios in a non-native way. Indeed, ATL is not a dedicated composition language and does not categorize rules on merging and transformation classes. It used *matched* rules to specify the transformation behavior in a declarative way; while *lazy* and *called* rules allow defining imperative transformations.

We have proposed to employ graph transformations to perform the weaving of the traceability aspect. For that, we use the ATL EMF-specific injector/extractor[1] to generate the corresponding model of an ATL concrete specification. Thereafter, a specific graph transformation unit is applied on the resulting model to weave the traces generation patterns. In the following sections, we detailed the main graph transformation rules which constitute this unit. The Henshin project [17] is used for specifying these graph transformations. We have to notice that the rule declaration in the Henshin formalism does not explicit the description of the left and right hand sides. Instead, it is based on the following stereotypes to depict the rule application semantic:

- *preserve*: all elements (edges or nodes) labeled with this stereotype must be sought to enable the rule application. Furthermore, those elements will be copied in the resulting graph.
- *create*: it is used to describe the new elements to be added to the graph.
- *delete*: it references the elements to be removed from the graph.
- *require*: this stereotype allows expressing of the conditions necessary for the rule application.
- forbid: by contrast, the presence of such a pattern prohibits the rule application.

1) Trace model declaration: The purpose of the rule depicted in Fig. 4 is to declare the trace model as an additional output of the module to trace. This model conforms to our generic traceability metamodel (cf. Section III-B) which corresponds to the other created node.

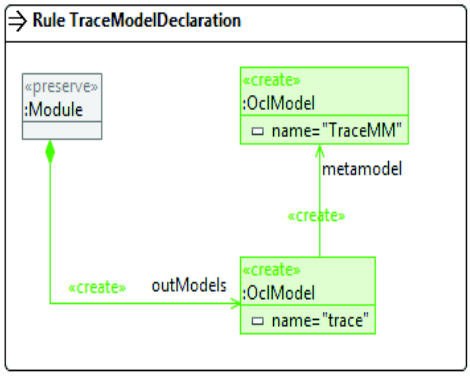

Fig. 4. Trace model declaration rule

2) Trace ATL rules: We have presented that ATL does not categorize rules on merging and transformation rules. Therefore, the user assistance is required to resolve the rule category in a composition scenario. For that, we have augmented the *rule* concept in the ATL abstract syntax with the *type* attribute. This attribute admits two values: *Merge* and *Transform* which are assigned manually depending on the intension of defining the rule. Note that we have experienced an automatic resolution of the rule category based on the number of input patterns. Actually, we consider rules with two input elements as simulating the merging behavior and the others as transformation rules. However, this hypothesis does not support a realistic selection, since a rule with two inputs may encapsulates a transformation behavior. Hence, we argue that even if this annotation mechanism is time consuming but it allows generating *trusted* [7] traces.

The Fig. 5 describes the graph transformation rule that allows tracing an ATL merge rule. Keeping track of such a

[1] See https://wiki.eclipse.org/ATL/Developer_Guide

rule consists of declaring the traceability link that captures the relationship between the contributing model elements and the target ones as being an extra output. Indeed, this graph transformation rule looks for an ATL rule annotated with the *Merge* type. Subsequently, it creates a new *SimpleOutPatternElement* node of type *MergingLink* that refers to the traceability link to be generated. The tracing of transformation rules is performed using a similar mechanism.

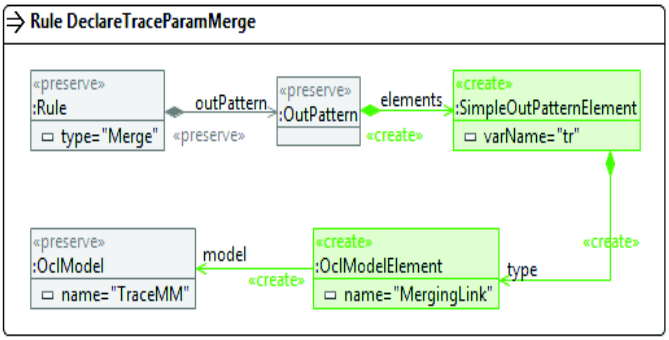

Fig. 5. Declaration of the traceability parameter for merge rules

In addition, we define additional transformation rules to connect traces to the model elements they link. These graph transformation rules look for the *InPattern* elements, *parameters* and *OutPattern* elements, then each one is assigned as a *left*, *right* or *targets* reference of the link generated by the current rule (which produces to the selected element). As an example, the rule depicted in Fig. 6 allows the assignment of *targets* references. It adds the selected *OutPattern* element as a further target of the trace link (referenced by the *tr* element) through the created *ExpressionStat* node.

3) Trace links nesting: During the execution of the resulting ATL module, the application of the pervious rules allows the generation of trace links while producing the target model elements. The links nesting will be closely modeled on the

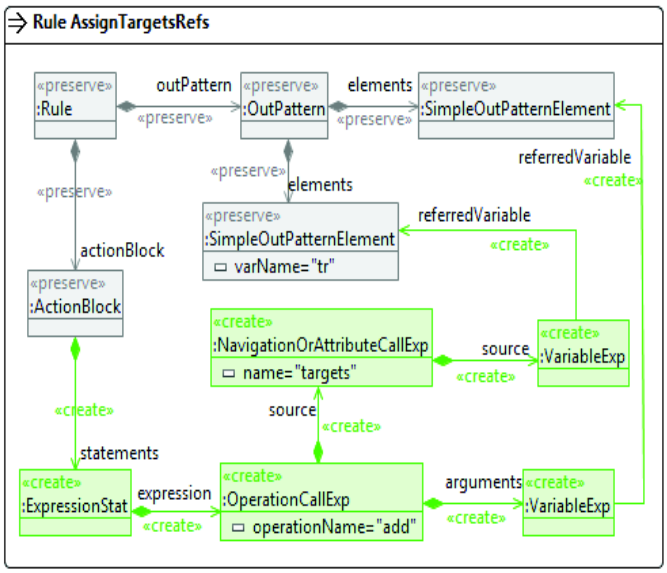

Fig. 6. Assignment of targets references

rules invocation sequence. Basically, we catch every call of a rule; then the traceability element created by the calling rule will be assigned as being a parent of the link generated by the called one. In ATL, the rule call can be performed with two mechanisms: an explicit call of *lazy* and *called* rules and an implicit call of *matched* rules through the use of the *resolveTemp* operation.

In the former case, we allow *called* and *lazy* rules to access the traceability element generated by the calling rule and assign it as parent of their trace link. This parent must be passed as parameter in the call expression. The rule depicted in Fig. 7 inserts the code responsible of this kind of nesting. This graph transformation rule matches a call of an operation which corresponds to a *called* rule name. Thereafter, it augments this call with a new parameter value referencing the parent link. Thus, the definition of the *called* rule has to be modified to take account of the newly passed value. For that, we add a new parameter (resp. an *InPattern* element in the case of *lazy* rules) of type *TraceLink* and create the *ExpressionStat* node that connects this parameter to the trace link generated by the rule that have been matched.

We have to notice that the same rule may be called several times, while the definition of the called rule has to be changed once. Actually, we use two rules to allow the explicit nesting. The first one adds the trace parent reference to the rule call expression and annotates the called rule. Thereafter, the second one browses all the annotated rules and changes their definition.

Regarding the explicit call, the rule responsible of nesting traces searches for a call of the *resolveTemp* operation. This ATL operation returns a target equivalent of a given source element. Considering that it results from the tracing of all *matched* rules, the production of additional outputs corresponding to the traceability data; those links can be returned as target equivalents of the element to resolve. Essentially, the nesting mechanism adds two statements. The first one copies the original call of the *resolveTemp* operation. The other statement assigns the selected trace equivalent to be a parent of the sibling trace link of the resolved source element. This filtering is based on the second parameter of *resolveTemp* that encodes the name of the target pattern element.

4) Context assignment: The use of generic purpose traceability metamodel proves advantages to support the reusability task. However, traceability data has to bring interesting information with respect to the traceability scenario. For that, the generic traceability metamodel need to be augmented with an extensibility mechanism to allow defining relevant expressiveness data regarding the traceability point of view and the models to compose specifications. The *context* and *contextAttribute* concepts defined in our traceability metamodel deal with this task. Basically, the definition of a context attribute is tied to the additional information to be appended to a specific set of traces, while the context concept is a well thought out combination of attributes.

In order to assign further information to trace links, the user has to provide the structuring of data to be appended using the proposed extensibility mechanism. Thereafter, all the possible contexts have to be generated. We define two ATL rules that support this task: the *createContextAttribute*

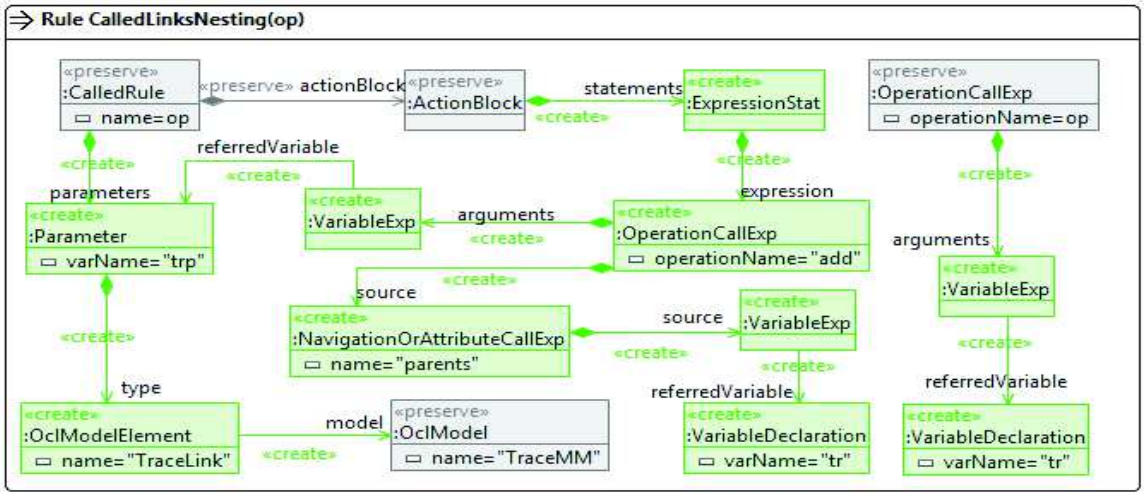

Fig. 7. Trace links nesting for explicit calls

rule (see Listing 1) that allows the production of a context attribute with respect to a given combination of the attribute *name* and *value*, thereafter, it binds the created attribute to its relevant context which is produced by the *createContext* rule (see Listing 2). Note that the possible values corresponding to a context attribute are either given by the user or automatically resolved through the definition of specific graph transformation rules. In either case, the context name is automatically assigned depending on the aggregated context attributes and can be viewed as a context key. The *createContext* rule is defined as *unique lazy* to prohibit the creation of multiple contexts with the same key.

```
rule createContextAttribute(atName:String, atValue:
    String, cName:String){
        to t:TraceMM!ContextAttribute()
        do{
                t.name<-atName;
                t.value<-atValue;
                t.owningContext<-thisModule.
                    createContext(cName);
        }
}
```

Listing 1. create context attribute for ATL

```
unique lazy rule createContext{
from cName:String
to t:TraceMM!Context()
        do{
        t.name<-cName;
        }
}
```

Listing 2. create context for ATL

Once all the contexts are generated, a graph transformation rule matches the declared traceability parameters; thereafter, it resolves the key of the context to be assigned to the selected parameter according to its connected contextual information. As a basic example, we propose to structure traces by the generating rule name. For this purpose, the context definition includes one context attribute which references the rule name contextual data. The declaration of contexts is performed using the graph transformation rule depicted in Fig.8. This rule looks for a *Rule* node, then, it creates the corresponding context through the invocation of the *createContextAttribute* rule. The passed arguments refer respectively to the context attribute name, the attribute value which corresponds to the selected rule name and the context name. The *appendTraceToContext* rule (Fig.9) allows connecting the traceability parameter to its context by calling the *createContext* operation, which returns the context identified by the selected rule name. Note that the *status* attribute has been added to the ATL abstract syntax to prohibit multiple application of these graph transformations to the same ATL rule (resp. traceability parameter).

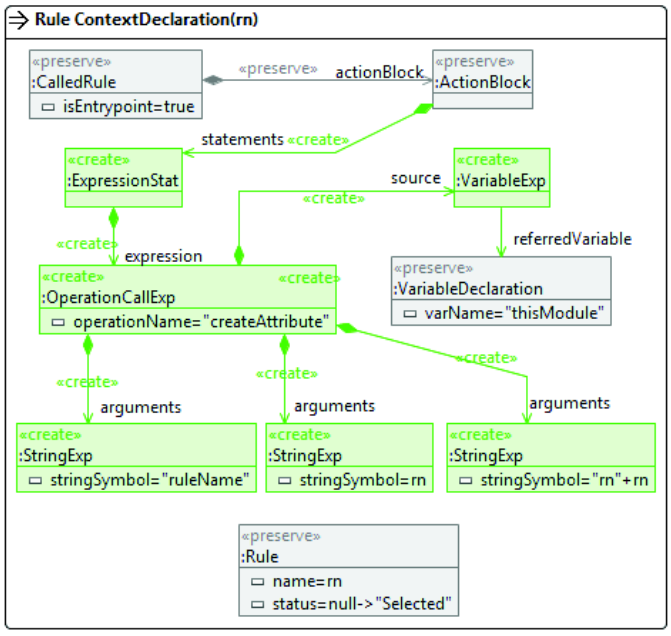

Fig. 8. The context declaration rule

V. Case Study

In this section, we illustrate the application of our traceability approach. The composition scenario we have chosen is the merging of two UML class diagrams into a VUML class diagram. We first briefly overview the VUML profile and

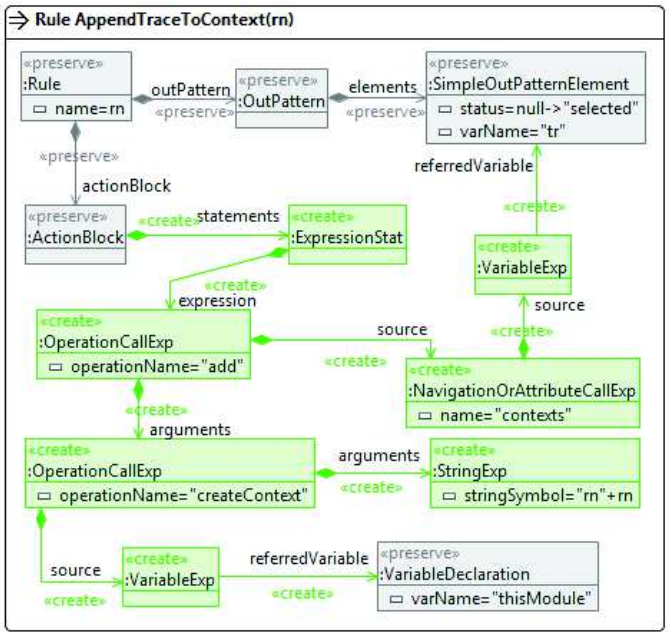

Fig. 9. Appending the traceability parameter to its context

introduce the case study. Thereafter, we present the results we obtain with our traceability framework.

A. The VUML profile

The VUML approach was developed to meet the needs of complex systems analysis and design according to various viewpoints [18]. In VUML, a viewpoint represents the perspective from which a given actor interacts with the system. In other words, at the requirements analysis step, a viewpoint expresses requirements and needs of one actor. At the design step, VUML extends the UML language by adding the concept of multiview class which is composed of a base class (shared by all viewpoints), and a set of view classes (extensions of the base class), each view class being specific of a given viewpoint. An actor's viewpoint on the system is expressed by a set of UML diagrams obtained by focusing on the relation of this actor with the system. A process was defined for VUML [19]. It begins by defining, for each actor, the use case diagram related to its needs, and then scenarios are modeled by means of sequence diagrams. Finally, static class diagrams are defined. Once all viewpoints are modeled, diagrams of the same type are merged into VUML diagrams, in order to express all viewpoints on a unique diagram, while preserving the possibility of selective access. VUMLs semantics is described by a metamodel, a set of well-formed rules expressed in OCL, and a set of textual descriptions in natural language.

B. Case study

In what follows, we illustrate the results of our traceability framework through the composition of two class diagrams that have been extracted from the Course Management System (CMS) [19]. The CMS is used by different users. It allows distant students to apply for courses, access related documentation (slides, web pages, text...), solve/create exercises, communicate with teachers and take exams. It allows teachers to edit their own courses, plan learning experiences and units of work, and record student assessments. The CMS is managed by an administrator whose job consists in recording students and managing resources.

According to the VUML process, the CMS is designed through a set of viewpoints (Student, Teacher and Manager). For each viewpoint, a set of UML diagrams (class diagrams, state machines, sequence diagrams...) are produced. The scenario we propose to trace is the composition of class diagrams. Fig.10 shows an excerpt of the class diagram of the Student's viewpoint focusing on the *Course* class, while Fig.11 depicts the class diagram that model the Teacher's requirements.

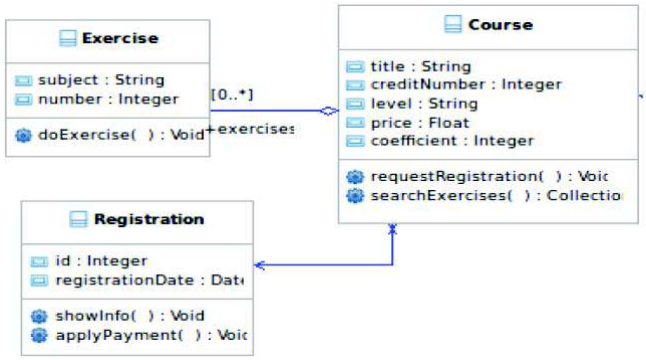

Fig. 10. Excerpt of the class diagram of the Student's viewpoint

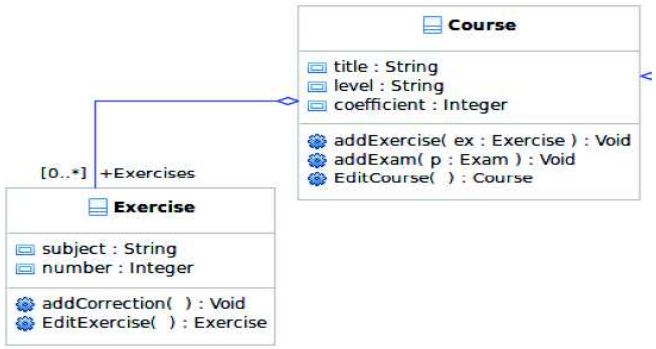

Fig. 11. Excerpt of the class diagram of the Teacher's viewpoint

These viewpoint models are composed to produce a VUML model. Fig.12 depicts the VUML class diagram resulting from the composition of the two class diagrams presented above. Classes appearing in both viewpoint models, with the same name and with different properties (attributes, operations, associations...), are merged as single multiview classes (*Course* and *Exercise* classes). Properties of the class *Course* that are shared by the two considered viewpoints have been put into the class stereotyped by *base*; properties that are specific of one viewpoint have been put into classes stereotyped by *view*.

The composition of viewpoint models requires a traceability mechanism for manifold applications. Actually, traces can be used to validate the consistency and the completeness of the resulting VUML model. Besides, they exposes the interactions between the managed models and provides means to support the propagation of changes occurred in viewpoints.

An extract of the ATL specification that allows performing the composition scenario is given in Listing 3, while Listing 4 depicts the modification resulting from the weaving of our traceability aspect.

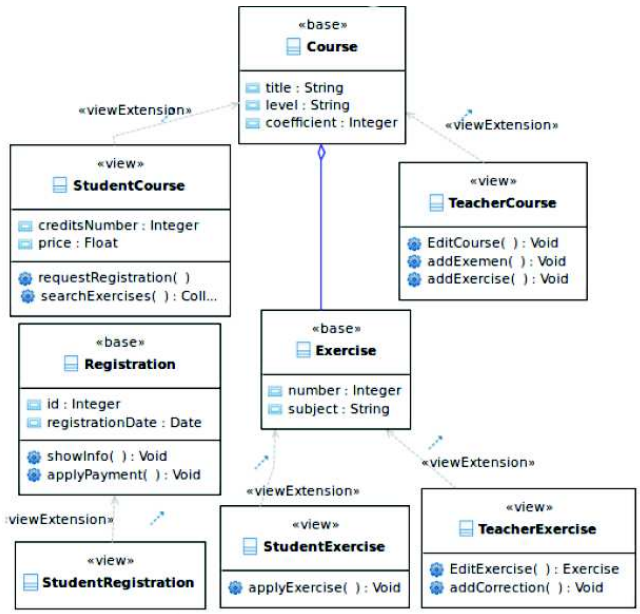

Fig. 12. Excerpt of the VUML class diagram of CMS system

```
1  module VUMLComp;
2  create VUML:UML2 from  MPV1:UML2, MPV2:UML2, PRO:UML2;
3  ...
4  entrypoint rule CreateCorrespModel(){
5  to cm : UML2!Package (name <- 'VUMLModel')
6  do{
7  ...
8  }
9  }
10 rule ClassSimilarity {
11 from l : UML2!Class, r : UML2!Class
12 (not(l.match(r)) and l.name=r.name and l.isLeft() and r.
       isRight())
13 to t : UML2!Class , v1 : UML2!Class, v2 : UML2!Class
14 do{...
15 for (it in l.getAttributes()){
16 if(not(it.getRightCorresp().oclIsUndefined())){
17 t.ownedAttribute <- thisModule.MergeAttributes(it,it.
       getRightCorresp());}
18 else{
19 v1.ownedAttribute<-thisModule.TransfomLAttributes(it);}
20 }
21 for(it in r.getAttributes()){
22 if(it.getLeftCorresp().oclIsUndefined()){
23 v2.ownedAttribute<-thisModule.TransfomRAttributes(it);}
24 }
25 ...}
26 }
27 lazy rule MergeAttributes{
28 from l:UML2!Property, r:UML2!Property
29 to t:UML2!Property
30 do{
31 t.name<-l.name;
32 t.type<-thisModule.resolveTemp(Tuple{l=l.type,r=r.type},'t')
       ;}
33 }
34 rule MergePrimitiveTypes{
35 from l : UML2!PrimitiveType, r: UML2!PrimitiveType (l.isLeft
       () and r.isRight() and l.match(r))
36 to t : UML2!PrimitiveType()
37 do{
38 t.name<-l.name;
39 thisModule.VUMLModel.packagedElement<-t;}
40 }
```

Listing 3. Extract of the initial VUML composition module

```
1  module VUMLComp;
2  create VUML : UML2, trace:TraceMM from  MPV1 : UML2, MPV2 :
       UML2, PRO : UML2;
3  ..
4  helper def : element : OclAny = OclUndefined;
5  entrypoint rule CreateCorrespModel(trp:TraceMM!TraceLink){
6  to cm : UML2!Package (name <- 'VUMLModel'),tr:TraceMM!
       TransformationLink()
7  do {
8  ....
9  tr.targets.add(cm);
10 thisModule.createContextAttribute('ruleName', '
       CreateCorrespondenceModel', '
       rnCreateCorrespondenceModel');
11 thisModule.createContextAttribute('ruleName', '
       ClassSimilarity', 'rnClassSimilarity');
12 thisModule.createContextAttribute('ruleName', '
       MergeAttributes', 'rnMergeAttributes');
13 thisModule.createContextAttribute('ruleName', '
       TransfomLAttributes', 'rnTransfomLAttributes');
14 thisModule.createContextAttribute('ruleName', '
       MergePrimitiveTypes', 'rnMergePrimitiveTypes')
15 ...
16 tr.contexts.add(thisModule.createContext('
       rnCreateCorrespondenceModel'));}
17 }
18 rule ClassSimilarity {
19 from l : UML2!Class, r : UML2!Class
20 (not(l.match(r)) and l.name=r.name and l.isLeft() and r.
       isRight())
21 to t : UML2!Class , v1 : UML2!Class, v2 : UML2!Class , tr:
       TraceMM!MergingLink()
22 do{
23 ...
24 for (it in l.getAttributes()){
25 if(not(it.getRightCorresp().oclIsUndefined())){
26 t.ownedAttribute <- thisModule.MergeAttributes(it,it.
       getRightCorresp(),tr);}
27 else{
28 v1.ownedAttribute<-thisModule.TransfomLAttributes(it,tr);}
29 }
30 for(it in r.getAttributes()){
31 if(it.getLeftCorresp().oclIsUndefined()){
32 v2.ownedAttribute<-thisModule.TransfomRAttributes(it,tr);}
33 }
34 tr.left<-l;
35 tr.right<-r;
36 tr.targets.add(t);
37 tr.targets.add(v1);
38 tr.targets.add(v2);
39 tr.contexts.add(thisModule.createContext('rnClassSimilarity
       '));}
40 }
41 lazy rule MergeAttributes{
42 from l:UML2!Property, r:UML2!Property, trp:TraceMM!TraceLink
43 to t:UML2!Property, tr:TraceMM!MergingLink()
44 do{
45 t.name<-l.name;
46 thisModule.element<-Tuple{l=l.type,r=r.type};
47 t.type<-thisModule.resolveTemp(thisModule.element,'t');
48 tr.parents.add(thisModule.resolveTemp(thisModule.element,'tr
       '));
49 tr.left<-l;
50 tr.right<-r;
51 tr.targets.add(t);
52 tr.parents.add(trp);
53 tr.contexts.add(thisModule.createContext('rnMergeAttributes
       '));}
54 }
55 rule MergePrimitiveTypes{
56 from l : UML2!PrimitiveType, r: UML2!PrimitiveType (l.isLeft
       () and r.isRight() and l.match(r))
57 to t : UML2!PrimitiveType() , tr:TraceMM!MergingLink()
58 do{
59 t.name<-l.name;
60 thisModule.VUMLModel.packagedElement<-t;
61 tr.left<-l;
62 tr.right<-r;
63 tr.targets.add(t);
64 tr.contexts.add(thisModule.createContext('
       rnMergePrimitiveTypes'));}
65 }
```

Listing 4. Extract of the resulting composition module with traces generation

As a result of applying the first rule of the traceability weaving unit, the trace model is declared as an additional target model of the VUML composition module (Listing 4: line 2). Thereafter, depending on the rule type (*Merge* or *Transform*), the traceability parameter is declared as another target parameter and the traceability information is assigned to it through the application of the relevant graph transformations (Listing 4: lines 21,34-38). During the execution of the composition module, these added parameters allow the generation of trace links that capture correspondences between the source model elements and the targets ones. Besides, in order to represent multi-scaled trace model, traces thus generated, will be nested with respect to the rules application sequence. Indeed, the graph transformation responsible of catching explicit calls, searches a call for a *lazy* rule (Listing 3: line 17). Then, it assigns the traceability element to be a child of the trace link created by the called rule (Listing 4: lines 26,42,52). As for the implicit call, the relevant links nesting rule searches for a call of the *resolveTemp* operation (Listing 3: line 32); and divides its return to trace model elements and target elements. The first sub-set is used to copy the original call of the called operation (Listing 4: line 47), while the traceability element is assigned as a parent of current trace link (Listing 4: line 48).

Recalling from Section IV-B4, the context concept is used to express highly configurable and semantically rich trace models through the assignment of further information to traces. To illustrate this augmentation mechanism, we have proposed two graph transformations that allow structuring traces according to their generating rules. The first one declares the possible contexts (Listing 4: lines: 10-14);while the second rule connects the trace link to its corresponding context (Listing 4: line: 39).

Fig.13 gives an extract of the generated trace model. Note that we have used the Emf2gv project[1] to provide a user friendly representation of traces. Trace links are presented with green rectangles and contexts with blue ones. The red, green and blue lines represent respectively the *left*, *right* and *targets* references. Dashed lines connect traces to their contexts. The trace links nesting is presented with solid lines.

The generated trace model is conforming to our composition traceability metamodel and represents how viewpoint elements contribute on the production of the VUML model. It contains two types of trace links that are generated with respect of the user's intensions. For instance, the root merging link connects the *Course* classes corresponding respectively to the *Teacher* and *Student* viewpoints to the target elements referencing the base class and the two generated views (*StudentCourse* and *TeacherCourse*).The Fig.13 depicts also a transformation link that represents the transition from the *creditNumber* attribute belonging to the *Course* class to its equivalent in the view class *StudentCourse*. Besides, the trace model includes further information to express semantically rich traces. Indeed, the user can visualize the rule that generates each trace link thanks to the use of the context concept.

VI. Related works

Jouault [15] presented an approach to trace model transformations written in the ATL language. This work was considered as a basis for several approaches addressing the transformation traceability management. The author proposed to generate the trace model in the same way other target models are generated. Actually, the code responsible of producing this extra output is inserted using a HOT named *TraceAdder*. This mechanism deals with the scalability challenge, since it allows automating the trace model generation task. Nevertheless, the definition of this HOT is closely tied to the proposed traceability metamodel which is very simple to express configurable traces. Moreover, the use of a practical metamodel requires a laborious and complex adaptation of the HOT *TraceAdder*.

In [20], the authors provide a traceability framework for the Kermeta language. The imperative character of such transformations makes them difficult to be analyzed. Indeed, they proposed to manually add the traces generation code. Their traceability metamodel expanded the one proposed in [15] by structuring traces into *steps*. This concept allows a basic configuration to the trace model. Besides, the manual encoding of the generation concern allows the user to trace the required elements, but this is to the detriment of scalability.

Aspect Oriented Programming (AOP) provides means to address the traceability management issues. Essentially, the traceability concern is encapsulated in a reusable aspect. Its application allows inserting the traces generation code automatically into the transformation specification. Within this scope, Amar et al. [21] proposed a traceability framework based on AOP to trace imperative transformations written in JAVA. The framework defines categories of traceable operations and their respective poincuts. Besides, the programmer can define new custom categories or restrict the predefined ones in order to take into account all the operations to trace. On the other hand, the authors apply the *composite* design pattern and the *LinkType* concept in view of expressing configurable traces.

Grammel and Kastenholz [22] have defined a generic framework to augment transformation supports with a traceability mechanism. The authors do not treat a specific transformation language, they provide a generic interface which involves specific connectors for various transformation supports instead. The augmentation is based on a generic traceability metamodel extensible with facets to express highly configurable trace models. Regarding the traces generation, the authors describe two mechanisms: transforming the implicit trace model to conform to their generic metamodel or capturing traces using AOP.

The aforementioned approaches can be used to keep track of the model composition operation. However, the fact that they disregard the composition process and intensions prevents them to express configurable and interesting traces. Against this context, our approach takes advantages of the existing solutions while focusing on the composition operation. We proposed to use an AOM approach to generate trace links. This mechanism allows encapsulating the traceability concern in a reusable aspect that automatically weaves the traces generation patterns. On the other hand, the abstraction of the specification to trace through its corresponding model provides support to master the composition language's features such as: the concrete syntax nature (textual, graphical) or the resolution of the rule category (merge or transformation).

As for the configuration challenge, we use a generic

[1] See http://sourceforge.net/projects/emf2gv

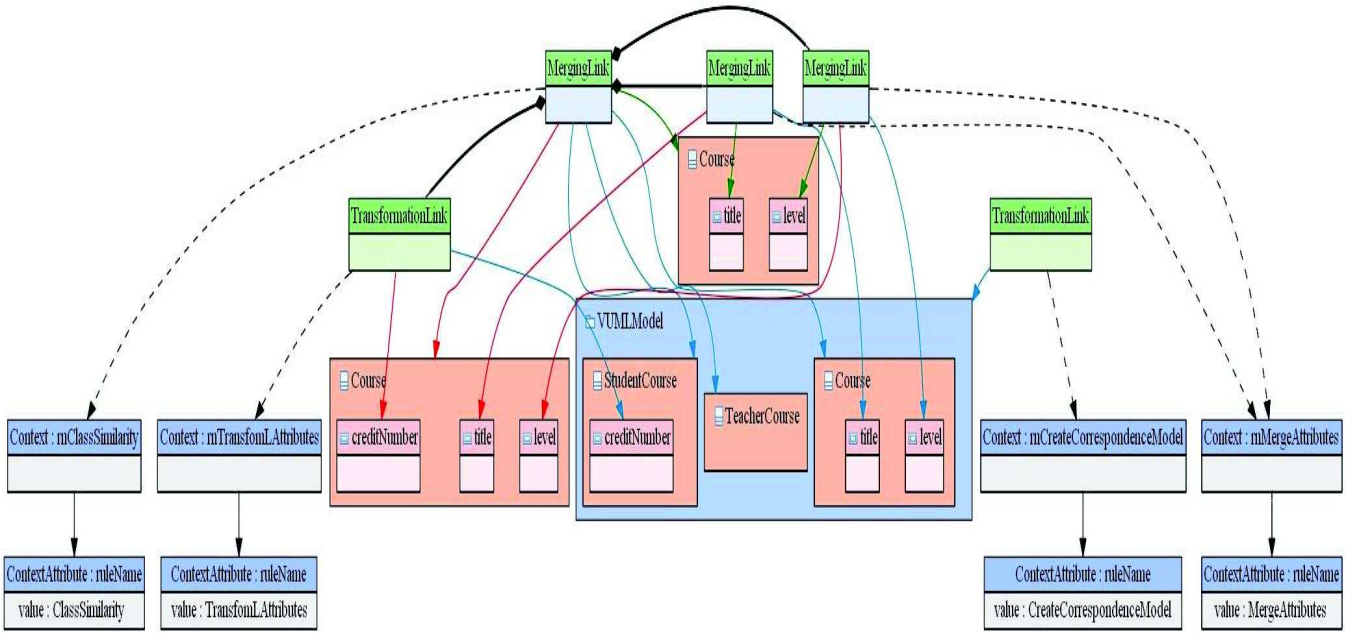

Fig. 13. Extract of the generated trace model

traceability metamodel to express reusable trace models. This metamodel was designed with respect to the typical composition process. Indeed, it categorizes traces on two sub-sets: merging links and transformation links. This allows expressing the composition relationships kinds in a trivial manner and will guide the reuse of traces. In addition, the *context* concept provides us with the mechanism to express highly configurable trace links. Actually, it allows the assignment of valuable information to the generated traces depending on the traceability point of view and specifications of managed models.

VII. Conclusion and Future Work

We proposed in this paper a graph transformation based approach to trace the model composition operation. Our objective is to define a purposed traceability approach that generates valuable traces for this particular operation, in an automatic, reusable and configurable manner. For that, we considered the traceability as being a cross-cutting concern. The weaving of the traces generation patterns is performed using graph transformations rules. This mechanism allows us to master the specificities of model composition approaches. Besides, we use a generic metamodel to express semantically rich traces.

We are currently working on a generic framework that allows defining the traceability aspect regardless the composition language. This framework will be augmented with a specialization mechanism for suitable application for a given language. Furthermore, we are planning to explore the possible reuses of traces. Our major intension is to optimize composition chains by using the previous generated links while executing a given step.

References

[1] A. Anwar, A. Benelallam, M. Nassar, and B. Coulette, "A graphical specification of model composition with triple graph grammars," in Model-Based Methodologies for Pervasive and Embedded Software. Springer, 2013, pp. 1–18.

[2] G. Rozenberg and H. Ehrig, Handbook of graph grammars and computing by graph transformation. World Scientific Singapore, 1997, vol. 1.

[3] J. Radatz, A. Geraci, and F. Katki, "Ieee standard glossary of software engineering terminology," IEEE Std, vol. 610121990, p. 121990, 1990.

[4] N. Drivalos, R. F. Paige, K. J. Fernandes, and D. S. Kolovos, "Towards rigorously defined model-to-model traceability," in ECMDA Traceability Workshop (ECMDA-TW), 2008, pp. 17–26.

[5] B. Grammel and K. Voigt, "Foundations for a generic traceability framework in model-driven software engineering," in Proceedings of the ECMDA Traceability Workshop, 2009.

[6] N. Anquetil, B. Grammel, I. Galvao Lourenco da Silva, J. Noppen, S. Shakil Khan, H. Arboleda, A. Rashid, and A. Garcia, "Traceability for model driven, software product line engineering," 2008.

[7] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic, "The grand challenge of traceability (v1. 0)," in Software and Systems Traceability. Springer, 2012, pp. 343–409.

[8] Y. Laghouaouta, A. Anwar, and M. Nassar, "A traceability approach for model composition," in 2013 ACS International Conference on Computer Systems and Applications (AICCSA). IEEE, 2013, pp. 1–4.

[9] J. Bézivin, S. Bouzitouna, M. D. Del Fabro, M.-P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. F. Paige, "A canonical scheme for model composition," in Model Driven Architecture–Foundations and Applications. Springer, 2006, pp. 346–360.

[10] R. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-oriented approach to early design modelling," IEE Proceedings-Software, vol. 151, no. 4, pp. 173–185, 2004.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, Aspect-oriented programming. Springer, 1997.

[12] L. Lambers, H. Ehrig, and F. Orejas, "Conflict detection for graph transformation with negative application conditions," in Graph Transformations. Springer, 2006, pp. 61–76.

[13] F. Jouault and I. Kurtev, "Transforming models with atl," in Satellite Events at the MoDELS 2005 Conference. Springer, 2006, pp. 128–138.

[14] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Merging models with the epsilon merging language (eml)," in Model Driven Engineering Languages and Systems. Springer, 2006, pp. 215–229.

[15] F. Jouault, "Loosely coupled traceability for atl," in Proceedings of the European Conference on Model Driven Architecture (ECMDA)

workshop on traceability, Nuremberg, Germany, vol. 91. Citeseer, 2005.

[16] A. Yie and D. Wagelaar, "Advanced traceability for atl," in 1st International Workshop on Model Transformation with ATL, 2009, pp. 78–87.

[17] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place emf model transformations," in Model Driven Engineering Languages and Systems. Springer, 2010, pp. 121–135.

[18] M. Nassar, B. Coulette, X. Crégut, S. Ebersold, and A. Kriouile, "Towards a view based unified modeling language," in ICEIS (3), 2003, pp. 257–265.

[19] A. Anwar, S. Ebersold, B. Coulette, M. Nassar, and A. Kriouile, "A rule-driven approach for composing viewpoint-oriented models." Journal of Object Technology, vol. 9, no. 2, pp. 89–114, 2010.

[20] J.-R. Falleri, M. Huchard, C. Nebut et al., "Towards a traceability framework for model transformations in kermeta," in ECMDA-TW'06: ECMDA Traceability Workshop, 2006, pp. 31–40.

[21] B. Amar, H. Leblanc, and B. Coulette, "A traceability engine dedicated to model transformation for software engineering," in ECMDA Traceability Workshop (ECMDA-TW), 2008, pp. 7–16.

[22] B. Grammel and S. Kastenholz, "A generic traceability framework for facet-based traceability data extraction in model-driven software development," in Proceedings of the 6th ECMFA Traceability Workshop. ACM, 2010, pp. 7–14.